ΠΑΝΕΠΙΣΤΗΜΙΟ
ΙΩΑΝΝΙΝΩΝ

ΤΜΗΜΑ: ΜΗΧ. ΗΛ. ΥΠΟΛΟΓΙΣΤΩΝ & ΠΛΗΡΟΦΟΡΙΚΗΣ

ΑΝΑΦΟΡΑ 2ης ΕΡΓΑΣΤΗΡΙΑΚΗΣ ΑΣΚΗΣΗΣ
ΑΛΓΟΡΙΘΜΟΙ ΓΙΑ ΔΕΔΟΜΕΝΑ ΕΥΡΕΙΑΣ ΚΛΙΜΑΚΑΣ

Ντόντης Βασίλειος : 3300
Ταλαμάγκας Ζήσης-Προκόπιος : 3340

Ξεκινώντας το πρώτο εμπόδιο που πρέπει να αντιμετωπίσουμε είναι να δημιουργήσουμε μία ρουτίνα η οποία θα μπορεί να διαβάσει ένα csv αρχείο ακμών και να δημιουργεί ένα pandas DATAFRAME. Αρχικά κάνουμε το διάβασμα των δεδομένων μέσω της εντολής pd.read_csv(). Τρέχοντας μετά τα Max_Num_Links αποθηκεύουμε τα πρώτα 2 values σε 2 βοηθητικές λίστες(node1list, node2list) με τις οποίες θα δημιουργήσουμε το dataframe (pd.dataframe()). Τα values θα μας χρειαστούν επίσης στην κατασκευή των γειτονιών τσεκάροντας αν είναι μοναδικά με σκοπό να

κρατήσουμε όλους τους γείτονες από κάθε element στην ίδια θέση, αλλά και για να γεμίσουμε τον πίνακα node_names_list, που κρατάει όλες τις κορυφές του γραφήματος. Να σημειώσουμε επίσης, ότι για το στοιχείο που βρίσκεται στην θέση I του πίνακα node_names_list, αντιστοιχούν οι γείτονες που βρίσκονται στην θέση I του πίνακα neighbourhoodarray. Αφού έχουμε δημιουργήσει τον πίνακα με τις κορυφές, φτιάχνουμε το dataframe και στην συνέχει το γράφημα. Προσθέσαμε και λίγο κώδικα για την επιλογή layout του γραφήματος.

Για την επόμενη ρουτίνα add_random_edges_to_graph κοιτάμε να πάρουμε ένα τυχαίο αριθμό ο οποίος δείχνει ποια θέση του πίνακα node_name_list θα πάρουμε για κορυφή. Για παράδειγμα, αν ο τυχαίος αριθμός είναι το 1 θα πάρουμε την κορυφή που αντιστοιχεί στο node_names_list[1]. Έπειτα με το νούμερο που πήραμε θα πάρουμε άλλο ένα τυχαίο νούμερο το οποίο θα το συγκρίνουμε με την πιθανότητα δημιουργίας ακμής που δίνεται από τον χρήστη. Αν το τυχαίο νούμερο είναι μικρότερο από αυτό του χρήστη δημιουργούμε την ακμή – συνδέουμε τους κόμβους, δηλαδή τους κάνουμε γείτονες και ανανεώνουμε την λίστα των γειτόνων καθώς από αυτήν θα παίρνουμε πληροφορίες και σε άλλες συναρτήσεις. Εν τέλει προσθέτουμε τις ακμές στο γράφημα μέσω της add_edge και σχεδιάζουμε το γράφημα.

Προχωράμε στην ρουτίνα add_hamilton_cycle_to_graph. Ένας κύκλος hamilton πρέπει να περιλαμβάνει όλες τις κορυφές του γραφήματος. Για να το επιτευχθεί αυτό τσεκάραμε όλες τις κορυφές(τρέχοντας από 0 μέχρι len(node_names_list) με την σειρά αν ενώνονται μεταξύ τους και αν δεν ενώνοντας τις ενώναμε. Επίσης χρειάζεται να τσεκάρουμε τον πρώτο κόμβο με τον τελευταίο καθώς το for δεν μας κάλυπτε αυτήν την περίπτωση. Ανανεώνουμε πίνακα γειτόνων, προσθέτουμε ακμές στο γράφημα και δείχνουμε το γράφημα στον χρήστη.

Την υλοποίηση της use_nx_girvan_newman την κάναμε μέσω της έτοιμης συνάρτησης του network, girvan_newman(G).

Όσον αφορά την δική μας υλοποίηση του αλγορίθμου Girvan Newman, ξεκινάμε με την δημιουργία μιας δικής μας συνάρτησης, της compute_coherent_components, η οποία υπολογίζει τις συνεκτικές συνιστώσες του γραφήματος και γεμίζει έναν πίνακα communities με μια λίστα για κάθε community. Αυτό το κάναμε γιατί θεωρήσαμε απαραίτητο να ξέρουμε ποιες είναι οι κοινότητες και από ποια στοιχεία απαρτίζονται, πληροφορίες τις οποίες δεν μπορούσαμε να πάρουμε από την εντολή του networkx number_connected_components. Παρόλα αυτά η συνάρτηση δεν λειτουργεί βέλτιστα, για αυτό και μπορούνε να φανούνε διαφορές στους χρόνους εκτέλεσης της έτοιμης GV με την δικιά μας GV. Αφού υπολογίσουμε τις συνεκτικές συνιστώσες του γραφήματος, κρατάμε ένα αντίγραφο τον communities στον πίνακα fakecommunities, καθώς για να τρέξει σωστά ο κώδικας στην while που ακολουθεί αναγκαζόμαστε να κάνουμε clear το communities list. Βρίσκουμε τον GCC, κρατάμε τον αριθμό τον communities και παίρνουμε τα απαραίτητα δεδομένα για τους γείτονες του GCC. Φτιάχνουμε μια while που ελέγχει τον αριθμό των communities, αν δηλαδή γίνουν +1 οι κοινότητες θα βγούμε από την while, καθώς καταλαβαίνουμε ότι έγινε διαμέριση. Αρχικά τοποθετούμε τις ακμές σε ένα dataframe και φτιάχνουμε το γράφημα για τον GCC, επειδή το χρειαζόμαστε για να πάρουμε το BC value μέσω της nx.edge_betweenness_centrality. Στην συνέχεια, έχουμε μια counter μεταβλητή η οποία στην πρώτη εκτέλεση της while αποθηκεύει σε μια μεταβλητή LCgraph το υπάρχων γράφημα και προσθέτουμε στον πίνακα LCs(πίνακας με όλους τους GCCs) τον GCC. Συγκρίνουμε τα BCs, διαγράφουμε την ακμή, ανανεώνουμε το neighbourhoodarray και ξανακάνουμε υπολογισμό συνεκτικών συνιστωσών για να γίνει σωστός έλεγχος στην while. Έπειτα αφού ‘τελειώσει’ η διαμέριση, Συγκρίνουμε το

fakecommunities με το τωρινό communities για να πάρουμε τις 2 νέες κοινότητες και μέσω της nx_comm.modularity να υπολογίσουμε το modularity της διαμέρισης που θα μας χρειαστεί σε άλλη συνάρτηση, το τοποθετούμε στον πίνακα MODULARITIES που κρατάει στοιχεία για όλες τις αρθρωτότητες και τελειώνουμε. Κρατήσαμε εντός της while το αρχικό γράφημα καθώς πρέπει να μπει ως όρισμα στην εκτέλεση της nx_comm.modularity.

Για την συνάρτηση divisive_community_detection απλώς καλούμε την συνάρτηση one_shot_girvan_newman_for_communities2 η οποία κάνει ακριβώς την ίδια δουλειά με την girvan newman που αναλύσαμε νωρίτερα, απλά έχουμε μια for για να γίνει η λειτουργία αυτή όσες φορές θελήσει ο χρήστης και επιπλέον επιλέγονται για τον υπολογισμό του BC ένα ποσοστό των κορυφών ως πηγές. Στην λειτουργία αυτή μας βοηθάει η συνάρτηση nx.edge_betweenness_centrality_subset. Επιλέγουμε τυχαία τις κορυφές με αριθμό το ποσοστό που έδωσε ο χρήστης, που θα είναι πηγές.

Συνεχίζουμε με την συνάρτηση visualize_communities που είναι υπεύθυνη για την εμφάνιση του γραφήματος στον χρήστη, αλλά με κάθε κοινότητα να έχει διαφορετικά χρώματα. Αρχικά φτιάχνουμε λίστες με όλες τις ακμές, και επιπλέον θεωρούμε κοινότητα και αν μια κορυφή αποκοπεί από την διαμέριση και μείνει μόνη της, κάνουμε έναν βρόγχο. Αφού δημιουργήσουμε το dataframe, δημιουργούμε το γράφημα, παίρνουμε τις θέσεις των κορυφών με την βοήθεια της _layout(G) και μέσω της draw_networkx_nodes βάζουμε ένα τυχαίο χρώμα στις κορυφές της κάθε διαφορετικής κοινότητας. Τέλος κάνουμε χρήση της plt.show() για να εμφανιστεί το γράφημα στον χρήστη μετά τον χρωματισμό των κοινοτήτων. Να σημειωθεί εδώ, ότι αλλάξαμε λίγο την my_graph_plot_routine, της βάλαμε άλλο ένα όρισμα έτσι ώστε να ξέρουμε αν καλείται από αυτήν την συνάρτηση. Γιατί αν καλεσθεί από εδώ το plt.show() πρέπει να γίνει στο τέλος.

Επιπρόσθετα, στην determine_opt_community_structure, έχουμε ήδη υπολογίσει τις αρθρωτότητες σε κάθε διαμέριση, οπότε έχουμε έτοιμα τα δεδομένα, τυπώνουμε στον χρήστη την κοινότητα με την μεγαλύτερη αρθρωτότητα και τυπώνουμε επίσης και τις δυο κοινότητες που δημιουργήθηκαν από αυτήν. Του δίνουμε την επιλογή να τυπωθεί και η αρθρωτότητα μιας διαμέρισης της επιλογής του. Τέλος δημιουργούμε ένα ραβδόγραμμα με τις τιμές αρθρωτότητας και το εμφανίζουμε στην οθόνη του χρήστη, με την βοήθεια της συνάρτησης plot.bar.

Θα επεξηγήσουμε επίσης και την συνάρτηση reconstruct_graph, η οποία ξαναφτιάχνει το γράφημα και το εμφανίζει στον χρήστη. Την δημιουργήσαμε καθώς στις συναρτήσεις που κάνουν διαμέριση χρειαζόμαστε να δημιουργήσουμε γράφημα του GCC για τον υπολογισμό BC και άλλαζε το ολικό γράφημα.

Σχετικά με τον σχολιασμό των πειραμάτων μας, το αξιοσημείωτο είναι η διαφορά χρόνου εκτέλεσης του networkxGV και του δικού μας GV, το οποίο εξηγήσαμε ότι κάνει περισσότερο χρόνο λόγω της εκτέλεσης της δικής μας compute_coherent_components, αλλά μπορεί να καθυστερούνε και οι υπόλοιποι υπολογισμοί που κάνουμε. Όμως σίγουρα το μεγαλύτερο ποσοστό της καθυστέρησης οφείλεται στην compute_coherent_components. Δεν παρατηρήσαμε κάτι άλλο αξιοσημείωτο, ήταν τα αναμενόμενα αποτελέσματα.

Τέλος, με την πρόσθεση ακμών ή κύκλου Hamilton, παρατηρούμε ότι απαιτεί περισσότερα βήματα η διαμέριση με τον αλγόριθμο Girvan Newman, αποτέλεσμα λογικό καθώς μειώνεται σημαντικά ο αριθμός των γεφυρών, που συνήθως είναι αυτές με το μεγαλύτερο BC value και η αφαίρεσή τους τελειώνουν τον Girvan Newman.